Adaptive Mesh Refinement for Parallel Processors

by

Marsha J. Berger

Ultracomputer Note #130
December, 1987

Adaptive Mesh Refinement for Parallel Processors

by

Marsha J. Berger

Ultracomputer Note #130
December, 1987

This work was supported in part by AFOSR grant number 86-0148, DOE grant number DEAC0276-ER03077-V and DARPA grant F49620-87-C-0065.

---

[1] To appear in the Proceedings of the SIAM Conference on Parallel Processing, Los Angeles, California, December 1987.

Adaptive Mesh Refinement for Parallel Processors

Marsha J. Berger

Courant Institute
New York University
251 Mercer St.
New York, N.Y. 10012

1. Introduction.

The goal of this work is to determine what kind of speedups are possible on a realistic application on today's parallel computers. The application I work on uses adaptive mesh refinement to solve the Euler equations for two-dimensional shock hydrodynamics. This adaptive mesh refinement strategy (henceforth AMR) was initially developed in [a]. More recently, it was been combined with the higher order Godunov methods of [b,c] to compute mach reflection for an oblique wedge. This combined code is 8000 lines of Fortran. It is used in several laboratories across the country. A typical run on a single processor of the Cray XMP takes 2 to 3 hours. The question is, what kind of speedups are possible using the other 3 processors?

One interesting question with adaptive mesh refinement, at least the kind of adaptivity AMR uses, concerns load balancing. The computational work in AMR changes dynamically in time, as well as being a function of space. Can the workload be efficiently divided so that all processors are kept busy, and no one waits idly for another CPU to finish? Another important question is the ease of parallel implementation. A major rewrite was out of the question, although certain subroutines had to be changed to be more amenable to multiprocessing. The parallel Fortran constructs available on different multiprocessors can make a huge practical difference. In particular, those of the Ultracomputer were much more convenient than those on the Cray.

In the next section, I very briefly describe the basic logic of AMR. I will only give the detail necessary for understanding my approach to parallelizing AMR. In section 3, I will review some alternative approaches to load balancing, including the one I use, called binary decomposition. Finally, section 4 discusses the parallelization of AMR, including how binary decomposition is applied to AMR, the parallel constructs used, and the numerical results and timings on two different machines - the Cray XMP and the Ultracomputer at NYU.

Acknowledgements. Some of this work was done in collaboration with Phil Colella. I would like to acknowledge the computer time on the XMP given to me by Cray. I thank Philippe De Forcrand for getting me started using the XMP, and Anne Greenbaum for helping me on the Ultracomputer. I thank Phil Colella and Scott Baden for many helpful discussions on all aspects of these problems. This work was

partially supported by grants from the AFOSR, DOE and DARPA.

2. The AMR Algorithm.

AMR uses nested grids with fine mesh spacing where more resolution is needed in the solution. This can be done recursively: several such nested levels of grid with increasingly finer mesh spacing may be used to achieve the desired resolution. These finer grids are simply *superimposed* on the underlying coarse grid in places where the resolution of the coarse grid is insufficient. By working with complete grids, instead of for example grid points, the data structures can be kept simple and regular. For a serial code, this makes vectorization possible, a property we would like to maintain in a parallel code as well.

The adaptive mesh refinement algorithm contains 4 basic components.

(1) An automatic error estimator determines the regions of high error. This determines where refinement is needed.

(2) An automatic grid generator creates fine grids patches covering the regions that need refinement.

(3) Simple data structures store the information describing the nested grids and manage the solution storage arrays.

(4) The integration strategy applies the same integrator to all rectangular grids.

The integration strategy needs to be described in more detail, since over 75% of the CPU time is spent integrating the grids. A key point of the algorithm is that when a grid is refined by a factor r in space, it is refined by the same factor r in time. This means that the mesh ratio $\frac{\Delta t}{\Delta x}$ is constant on all grids, so the same integration method is stable on all grids. This also means that the computational work is concentrated on the fine grids, where it should be. For every step on the coarse grid, r steps are taken on the fine grid, r^2 steps on the next finer level grids, etc.

Of course, the grids can not continue to be integrated independently of each other indefinitely. They interact in two ways. When the fine and coarse grids reach the same physical time, the fine grids update the underlying coarse grid points. This update step is a simple averaging procedure. If i,j are the indices of a coarse grid cell, and k,l are the indices of the lower left fine grid cell contained in the coarse grid cell, a conservative updating procedure is

$$u_{i,j}^{coarse} \leftarrow \frac{\sum_{n=1}^{r}\sum_{m=1}^{r} u_{k+m,l+n}^{fine}}{r^2}. \qquad (1)$$

The second interaction between grids happens at grid interfaces. Fine grids need "boundary" conditions that come from linear interpolation in space and time from the surrounding coarse grid. In order for the combined difference scheme to be conservative, the coarse grid needs to be modified correspondingly so that the difference equations at the coarse points immediately adjacent to a fine grid *see* the fine grid.

Since the grids are integrated independently, (and in fact, the coarse grid is updated first, before fine grid fluxes have even been calculated), this last requirement is implemented as a fix-up step after each fine grid step. This fixup pass takes the following form. First, a provisional coarse value is computed at the coarse points adjacent to fine grids,

$$\bar{u}_{i,j} = u^n_{i,j} + \frac{\Delta t}{\Delta x} D_+ F^{coarse} + \frac{\Delta t}{\Delta y} D_+ G^{coarse}. \tag{2}$$

This step provides values at the coarse grid points needed for the interpolation of the fine grid boundary values. When the fine grids are advanced, all the boundary fluxes are saved. The coarse grid is then modified so that the coarse grid difference scheme is fully conservative at the fine grid interface. For example, at a coarse grid point immediately to the left of a fine grid, the flux F across the right side of the cell must be modified to give

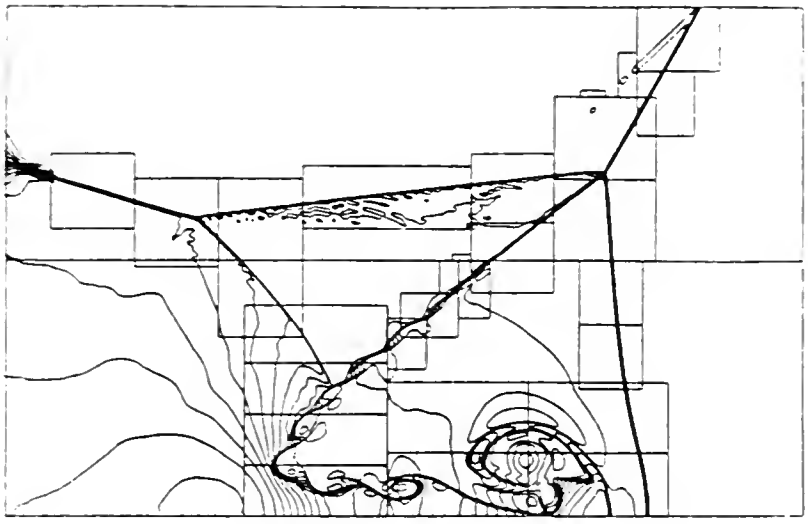
$$u^{n+1}_{i,j} = \bar{u}_{i,j} - \Delta t \left[F^{coarse}_{i+1/2,j} - \frac{\sum_r F^{fine}_{1/2,k}}{r} \right]. \tag{3}$$

The coarse grid keeps a list of coarse grid points that need adjustment. Eq. (3) can then be implemented as a scatter-gather operation on machines with such hardware. Although it is common to neglect boundary work in work estimates, in our initial implementation, boundary work took 40% of the CPU time and had to be rewritten. Complete details about these steps are found in [d].

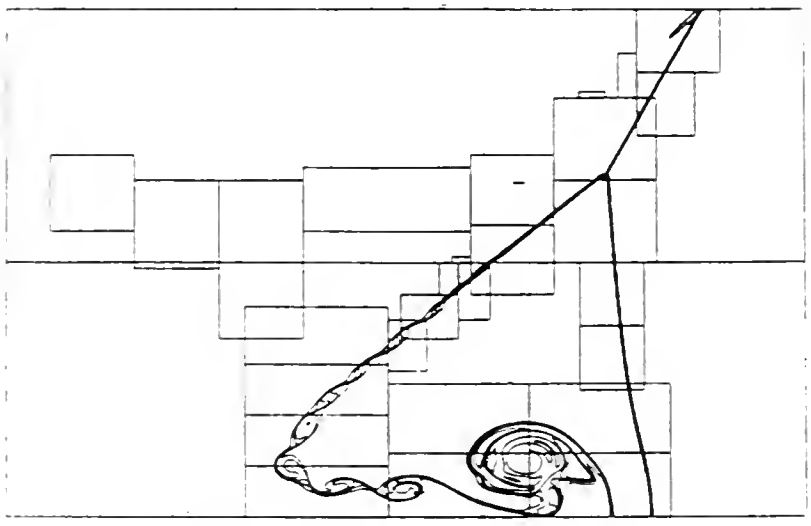
Figure 1 shows a typical calculation with this method, also taken from [d]. This particular run took 1 1/2 hours on the XMP. Although this is a well studied test problem, the use of adaptivity led to greater resolution than was previously possible, and a new feature in the solution, a Kelvin Helmholtz instability along the slip line, can be seen. The coarsest grid used only 100 by 20 cells, and 149 coarse grid time steps were taken. Three levels of grid refinement were used in this run. The grids were refined by a factor of 4 in each direction at each level. Table 1 summarizes where the computational time was spent.

grid integration	76.4%*
interpolation	13.1%*
error estimation	3.4%
grid updates	2.8%*
I/O	2.7%
grid generation	.6%
space management	.6%

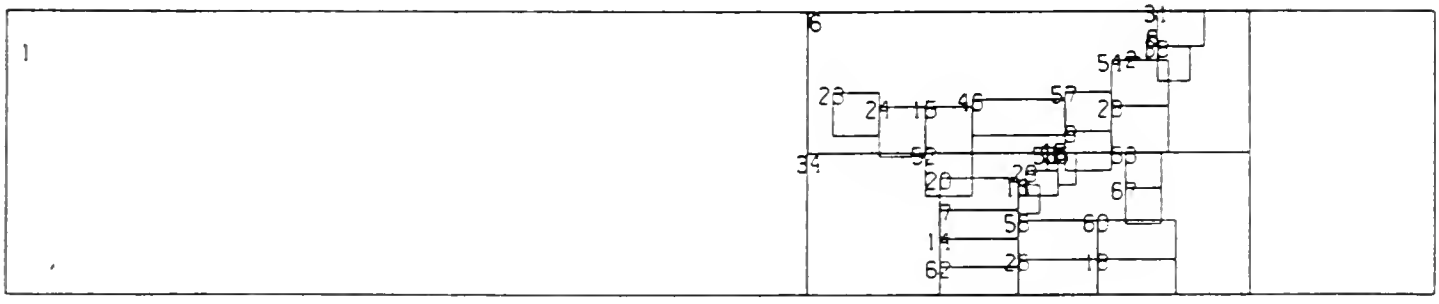
Table 1. Flowtrace information shows how the computational time is spent.



Density



Entropy



the grid hierarchy

Figure 1. Mach reflection calculation.

An important thing about these timings is that 92% of the runtime is accounted for in 3 (categories of) subroutines. In fact, 89% of the runtime can be accounted for in one subroutine. This is obviously the place to start the parallelization. The structure of this one subroutine is particularly simple. It is illustrated in Figure 2, since it forms the basis of the parallel processing of AMR.

Take 1 time step for all grids at a level.

Get next grid at this level
    interpolate boundary conditions
    integrate the grid
    save fluxes if necessary

Figure 2. 89% of the CPU time is spent in the subroutine that has the above logic.

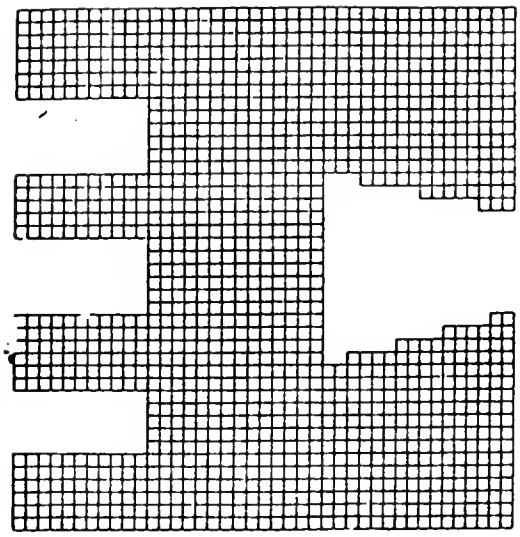
3. Load Balancing for Non-Uniform Computations.

Clearly the usual approach of dividing a domain into squares or rectangles of equal area will not balance the computational load across multiple processors. Since AMR already uses a type of "domain decomposition" to achieve an accurate solution at a minimum cost, it might at first appear that this decomposition could form the basis of the parallel processing of AMR. Different grids could be distributed to different processors. However, this would not balance the computational load, since there is no attempt *a priori* to create the same number of grids as processors, or to make the grids approximately the same size. Some other approach to load balancing is necessary.

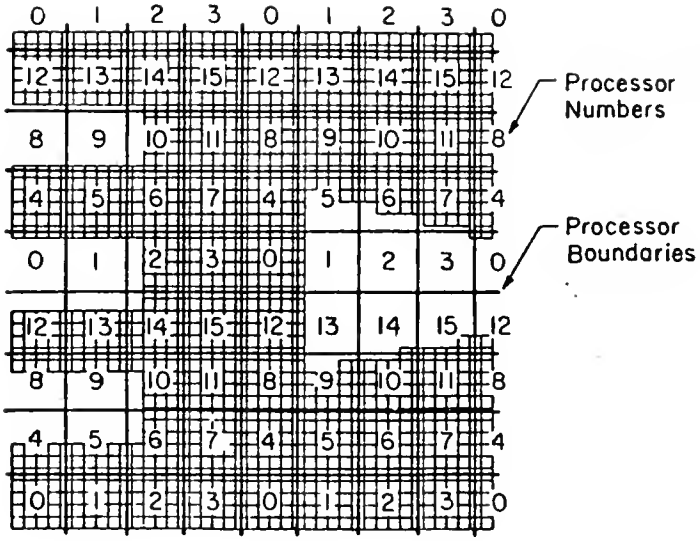
I will briefly summarize 4 approaches to load balancing non-uniform computational loads. I have divided them into the two categories of fine-grained and coarse grained decompositions. The finer the granularity (smaller the task size relative to the total amount of computational work per processor), the more chance the processors have of finishing their work at the same time, but the greater the overhead of creating, managing, and distributing the tasks. Other things being equal, a larger granularity is preferred.

The fine-grained approaches are a scattered decomposition, and a self-scheduling strategy. The coarse-grained approaches are simulated annealing and binary decomposition. I will illustrate them pictorially.

The scattered decomposition figures are taken from Morison and Otto [e]. Figure 3a shows a finite element mesh for a domain with an irregular boundary. To divide the work evenly among 16 processors, the domain is divided into many small pieces as if it were a regular domain. Some pieces will have more work than other pieces, which contain regions outside the problem domain. However, since each processor will receive many pieces (9 in this decomposition of Figure 3b), the hope is that the work load per processor evens out. My concern with this technique is that by dividing the work into many more pieces than

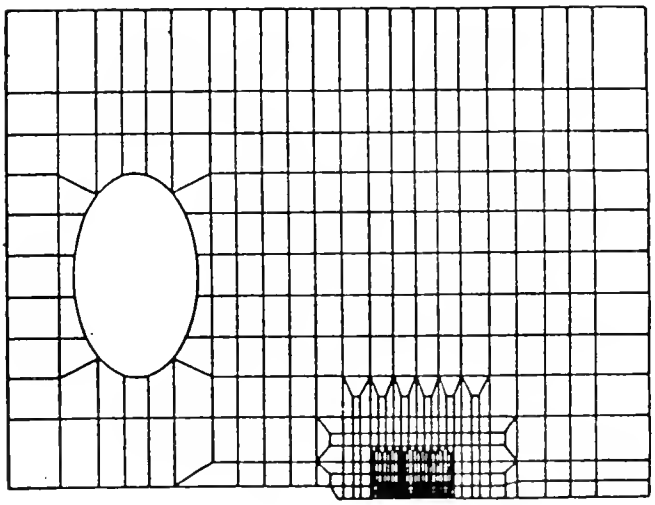


(a) computational mesh

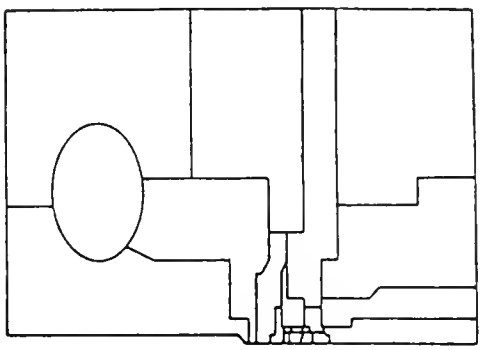


(b) the decomposition

Figure 3. Scattered decomposition.



(a) computational mesh



(b) the decomposition

(c) load distribution

Figure 4. Simulated annealing decomposition.

processors, the inter-processor communication, typically proportional to the length of boundary of the sub-domains, has been greatly magnified.

Self-scheduling strategies are typically used on shared memory multiprocessors. In this approach, a queue of small tasks is maintained, say by one of the processors. When requested, the work is distributed to available processors. The "Fetch and Add" primitive on the Ultracomputer was designed to prevent such a queue from being a serial bottleneck. Here again, the task size cannot be too small or the work of maintaining the queue and distributing the tasks will swamp the original computational work. Self-scheduling has been used by Greenbaum in [f] to do sparse back-substitution, and by Phil Colella in parallelizing a front tracking code using SLIC. In this latter work, the small tasks were single rows of the computational domain. Depending on how much of the front lay in the row, the work per row varied by up to 50%.

The coarser-grained approaches to load balancing generally use as many domains in the decompositions as there are processors. In the next approach, the load distribution problem is regarded as a discrete combinatorial optimization problem. The next step is to apply optimization techniques, such as the Monte Carlo method of simulated annealing. Figure 4 was taken from Flower, Otto and Salama [g]. Figure 4a again shows a finite element mesh for an irregular geometry. Figure 4b shows the domain decomposed using simulated annealing. Figure 4c shows the success of this approach in equidistributing the computational work over the 16 processors. An advantage to this approach is that the objective function can contain many complicated terms, if desired, for a complete description of the computational and communication complexity of the problem. Drawbacks, however, are that simulated annealing methods are very computational expensive, requiring hundreds of iterations to reach a solution. (The parallelization of the simulated annealing algorithm becomes a new computational problem by itself.) In addition, without additional work, the boundaries of each processors computational domain can be irregular, complicating the data structures needed to represent the decomposition.

The last method to be reviewed (and the one I use) is called binary decomposition. It is the most straightforward of the domain decompositions. The idea is to account for the workload in partitioning the domain, and to stick with simple shapes. In this approach, you have to have an *a priori* estimate of the computational work over the entire domain. The work can then be divided as equally in half as possible by a single vertical line, and the left and right halves given to two processors. If there are 4 processors, the decomposition continues recursively, using a horizontal line next, on each of the two halves. Figure 5, taken from [h], shows a sample binary decomposition for 16 processors. Figure 6, taken from [i], shows an application of binary decomposition to the solution of the incompressible Euler equations using a vortex method. In Baden's work, the measure of the computational work used was the number of vortices in each domain. If only vertical lines are used to partition the domain, you get strips. This has been used by Gropp in [j] in his earlier work on partitioning an adaptive mesh refinement program.

On the shared memory multiprocessors used in this work, the mapping of domains to processors is unimportant. However, some interesting theoretical properties of such mappings were studied in [h]. For

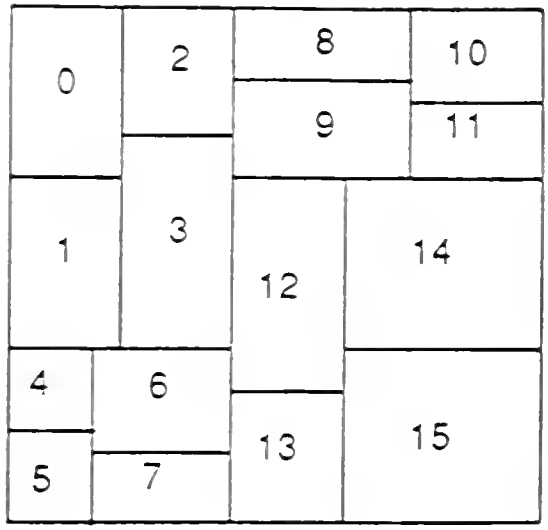


Figure 5. A binary decomposition for 16 processors.

example, consider the dual graph of the binary decomposition. (Regions sharing a common boundary segment have an edge connecting them in the dual graph). If this dual graph is mapped onto 4 nearest neighbor arrays, no fewer than 50% of the edges fall on edges of the machine graph, regardless of how skewed the partitioning is. This is important, since the edges that do not fall on machine graph edges incur higher communication costs. On an 8 nearest neighbor array, at least 79% of the edges fall on edges of the machine graph. Binary decompositions can also be mapped in a natural way onto a tree architecture. In this case, the communication cost is determined not by the longest boundary segment, but by the sum of such segments.

4. The Parallelization of AMR.

Since the binary decomposition partitions the domain into rectangles, it seems well suited for application to AMR, which also partitions space into rectangles. Of course, with AMR the partition must be a function of time. Whenever the grid hierarchy is changed, the partition must be changed. The decomposition is implemented by partitioning the grids in the grid hierarchy at a given time for assignment to different processors. (Note that this is not the same as distributing the original grids in the hierarchy themselves).

When I started this work, my original idea was to partition the entire grid structure, based on all grids in the grid hierarchy. This is illustrated in Figure 7a. This is the largest granularity possible. If the coarse and fine grids in the same region of space are owned by the same processor, the updating step of the algorithm does not incur inter-processor communication. Although this is theoretically preferable, it is not practical. Such a decomposition does not balance the work at each grid level, only the total work of all levels. Since coarse grids are not load balanced, for example, there should be no synchronization after coarse grids are integrated, before moving on to fine grids. This means that before any of the r integration steps on the fine grid, the boundary values have to be interpolated from the coarse grids. The stencil we use in the higher order Godunov method requires 4 points to the side. Typically, the refinement ratio $r=4$ as well.

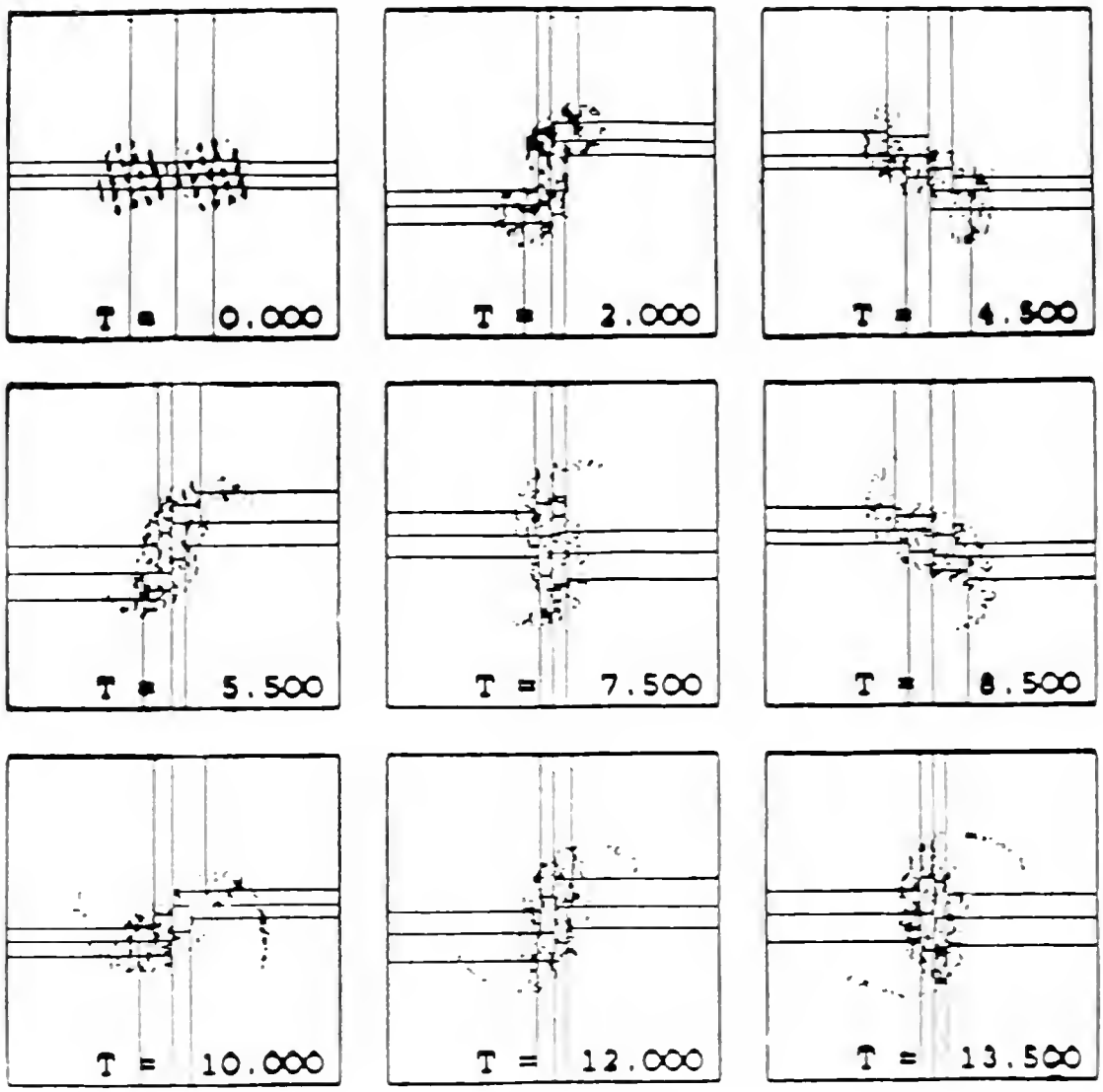


Figure 6. Binary decomposition applied to a vortex method.

This means 16 boundary values on each side of a grid must be set, or 32 in each direction. Although this is boundary work, this is not at all a negligible amount of extra work at each step.

The solution is to partition the grids by level. For example, if level 1 is just a simple rectangle, this gives the straightforward partitioning of figure 7b. The fine grids at level 2 are also partitioned by themselves. (Note that since most of the work is on the fine grids, the decomposition is very similar to the one in Figure 7a). As far as the updating step goes, for a refinement ratio of 4, one coarse point is updated by the average of 16 fine grid points. This update is performed every 4 fine grid time steps. Although at first thought this work is proportional to the area of the grids, in fact it is less time-consuming than the boundary work.

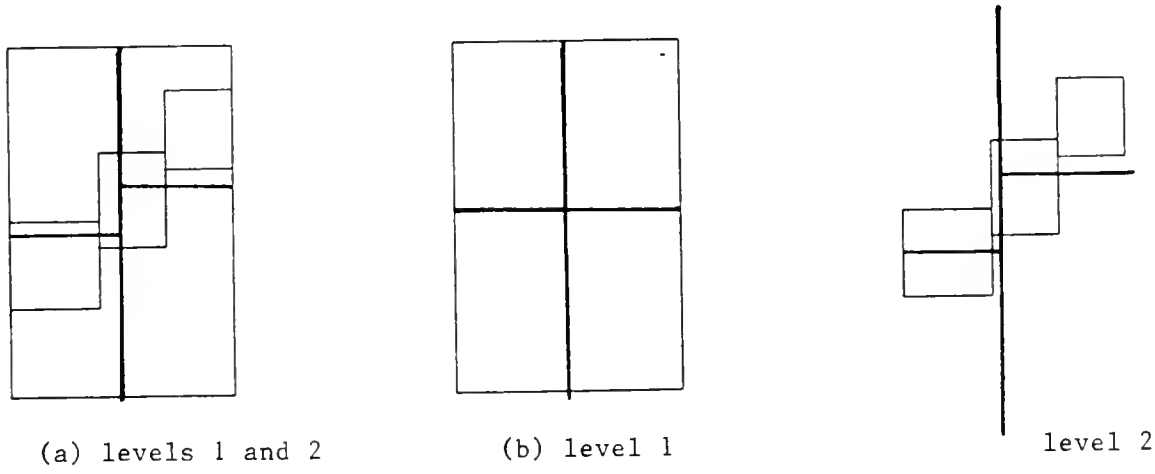


Figure 7. (a) Partitioning on the entire grid structure; (b) Partitioning by levels

Once the domain has been decomposed, grids in a given region are assigned a processor number. Only the owner processor may update a grid. Grids that span two such domains are subdivided, and each resulting grid is then assigned. Because of this division, there is some extra overhead associated with the decomposition, even if the program if run in serial mode. In my experiments, this serial degradation due to the extra grids is 3 - 5%.

The next step was the actual implementation using parallel Fortran constructs. AMR, like many if not most big scientific codes, contains chunks of computationally intensive, parallelizable code separated by long sections of serial code that don't take much time. The serial code does however account for most of the subroutines. The two machines I worked on, the XMP and the Ultracomputer, are quite different in their approach to parallel programs. Cray actually provides two mechanisms for parallelization. The main one is called macrotasking. With macrotasking, several CPU's are initialized for parallel threads of execution. Since the start-up time is high, the intent is to keep all the CPU's alive once they are initialized.

However, it is difficult to "turn them off" for the serial sections. The other mechanism for parallel processing is called microtasking. This has evolved quite a lot, and the intent for its final use is still not clear. Initially, it provided do loop level parallelism on a self-scheduling basis. There is no intent to provide a prespecified number of processors, nor give the user as much control over the CPU's in microtasking as in macrotasking. Although initially, subroutines could not be called within a microtasked loop, this restriction has been removed. However, microtasked loops cannot be nested.

In contrast, the Ultracomputer uses a "fork-join" model of parallelism. The user can explicitly initiate and terminate parallel section using a DOALL/ENDALL construct, for multiple CPU's executing the same code, or a PARBEGIN/PAREND construct for CPU's executing different sections of code. These constructs are limited to intra-procedural parallelism. However, unlike microtasking, they can be combined, on a self-scheduled basis, by nesting DOALL's.

This explicitly controllable parallel construct provides a particularly simple way to parallelize AMR. By adding 3 lines, the one subroutine accounting for 89% of the CPU time looks like this (cf. Figure 2).

```
Take 1 time step (in parallel) for all grids at a level.
  DOALL icpu = 1, ncpu
    Get next grid at this level
    if (icpu = grid_owner) then
        interpolate boundary conditions
        integrate the grid
        save fluxes if necessary
  ENDALL
```

Figure 8. This subroutine is parallelized by adding DOALL/ENDALL and one IF test.

If we only parallelize the one subroutine using 89% of the run time, perfect results would get a relative time of $89/4 + 11 = 331/4\%$, for a speedup of $S = 3.0$, and an efficiency $E = 75\%$. However, we get $S = 2.7$, which is an efficiency $E = 69\%$. This imbalance is due to 2 things:

(1) The fine grids cannot be bisected exactly in half. A fine grid is constrained in AMR to start and end at a coarse grid point (so that every 4 fine grid points is a potential dividing place).

(2) Perimeter (i.e., boundary interpolation) costs were not included in the work estimates.

Measurements show that the first problem causes 8-9% of the imbalance. The second problem, which is much more easily remediable, is responsible for only 2-3% of the imbalance.

For the next step, I parallelized the 5 subroutines that account for 98% of the CPU time. These next results use the Ultracomputer; the network and transmission lines to the XMP became too unreliable. Also,

these timings omit the I/O time to output the results, which is not done in parallel. Table 2 shows timings of the 5 subroutines as well as the total CPU time, for a clearer idea of how successful the parallelization is. The "mod. code" column refers to the serial code but with the grids subdividied as they need to be for parallel execution.

	CPU time in 5 subr.	total CPU time
orig. code, 1 CPU	28097	28412
mod. code, 1 CPU	28918	29281
4 CPU's	8041	8403
speedup over mod. code	28918/8041 = 3.60	29281/8403 = 3.4
speedup over orig. code	28097/8041 = 3.49	28412/8403 = 3.38

Table 2. Timing results of parallelization of 5 subroutines.

The load imbalance here is again approximately 10%. The efficiency in parallelizing the 5 subroutines is thus 90% when compared to the modified code (where the grids are subdivided, causing extra boundary work), and 87% when compared against the original code. The overall efficiency is then 87% over the modified code, and 84.5% over the original code. On the Cray, there is an additional approximately 2% penalty for the modified code over the original code, due to the shorter vector lengths in the subdivided grids.

5. Conclusions.

We have taken a realistic application, and achieved a speedup of 3.4 using 4 processors. It involved only minor changes to the existing serial code, which admittedly was readily amenable to a domain decomposition approach to parallel processing. However, the ease of use of the parallel constructs make a huge difference in the amount of work, and debugging effort necessary to achieve this speedup. Unfortunately, the Cray XMP models available today do not actually allow all 4 CPU's to be used on a routine basis. In fact, the job priority is often lower if more than one CPU is requested. When supercomputer multiprocessors are more readily available, more experimentation can be done.

6. References

[a] M. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential equations", J. Comp. Phys. 53, March, 1984.

[b] P. Colella, "Multi-dimensional Upwind Methods for Hyperbolic Conservation Laws", Lawrence Berkeley Laboratory Report LBL-17023, May, 1984. To appear in J. Comp. Phys.

[c] P. Woodward and P. Colella, "The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations", J. Comp. Phys. 54, April, 1984.

[d] M.J. Berger and P. Colella, "Local Adaptive Mesh Refinement for Shock Hydrodynamics", Lawrence Livermore Report UCRL-97196, Sept. 1987. To appear in J. Comp. Phys.

[e] R. Morison and S. Otto, "The Scattered Decomposition for Finite Elements", Caltech Report No. C^3P286, May 1985.

[f] A. Greenbaum, "Solving Sparse Triangular Linear Systems Using FORTRAN with Parallel Extensions on the NYU Ultracomputer Prototype". New York University Ultracomputer Note #99, April, 1986.

[g] J. Flower, S.W. Otto and M.C. Salama, "A Preprocessor for Irregular Finite Element Problems", Caltech Report No. C^3P292, June 1986.

[h] M.J. Berger and S.H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors", IEEE Trans. Comp. C-36, May, 1987.

[i] S.B. Baden, "Dynamic Load Balancing of a Vortex Calcuation Running on Multiprocessors", Lawrence Berkeley Laboratory Report LBL-22584, Dec. 1986.

[j] W.D. Gropp, "Local Uniform Mesh Refinement on Loosely-Coupled Parallel Processors", Yale University Computer Science Dept. Report RR-352, Dec. 1984.